Data Wrangling: Clean, Transform, Merge, Reshape

- After loading into a DataFrame, raw data needs to be pre-processed before used for data analysis
- Data from difference sources needs to be selected, combined, etc.
- Missing values needs to be filled in with other useable values
- Noise needs to be smoothed using binning methods
- Values may need to be normalized, mapped into a different range, encode/decoded, transformed into other format
- Tables may need to be merged, or with structure changed, etc.
- Pandas provide functions to perform all these tasks

In [ ]:
```
%matplotlib inline

from __future__ import division
from numpy.random import randn
import numpy as np
import os
import matplotlib.pyplot as plt
np.random.seed(12345)
plt.rc('figure', figsize=(10, 6))
from pandas import Series, DataFrame
import pandas
import pandas as pd
np.set_printoptions(precision=4, threshold=500)
pd.options.display.max_rows = 100
```

Combining and Merging Data Sets

- The merge() function combine rows from different DataFrames based on values in selected columns
  - This is Pandas equivalence to the relational equi-join operation
  - Both inner and outer joins are available.
  - view DataFrames as SQL tables
- Basic syntax: leftTable.merge(rightTable, otherParameters....)
- Result: rows from left and right tables are merged into one row in the result table if both rows have identical values in selected columns (or indices)

In [ ]:
```
df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                 'data1': range(7)})
df2 = DataFrame({'key': ['a', 'b', 'd'],
                 'data2': range(3)})
df1
```

In [ ]:
```
df2
```

Types of Relational Join

- Natural Join: Merge rows if they have identical values under columns with identical names
  - Ex: $R(A, B, C) \bowtie S(D, B, E) = RS(A, B, C, D, E)$
- Equi-Join: Merge rows if they have identical values under selected pairs of columns
  - Ex: $R(A, B, C) \bowtie_{R.A=S.D} S(D, B, E) = RS(A, B, C, D, B, E)$
- outer joins: Keep all rows in left or right or both table, even if they do not have matching rows in the other table. Fill missing values with NaN

In [ ]:
```
# Natural join (on the common column 'key')
pd.merge(df1, df2)
```

In [ ]:
```
pd.merge(df1, df2, on='key')
```

In [ ]:
```
df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                 'data1': range(7)})
df4 = DataFrame({'rkey': ['a', 'b', 'd'],
                 'data2': range(3)})
```

In [ ]:
```
# Equi-join on lkey = rkey
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

In [ ]:
```
# Outer natural join on common column 'key'
pd.merge(df1, df2, how='outer')
```

In [ ]:
```
df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                 'data1': range(6)})
df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
                 'data2': range(5)})
```

In [ ]:
```
df1
```

In [ ]:
```
df2
```

In [ ]:
```
# Left outer equi-join
pd.merge(df1, df2, on='key', how='left')
```

In [ ]:
```
# Natural join
pd.merge(df1, df2, how='inner')
```

In [ ]:
```
left = DataFrame({'key1': ['foo', 'foo', 'bar'],
                  'key2': ['one', 'two', 'one'],
                  'lval': [1, 2, 3]})
right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
                   'key2': ['one', 'one', 'one', 'two'],
                   'rval': [4, 5, 6, 7]})
```

In [ ]:
```
# outer equi-join on multiple columns
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

In [ ]:
```
pd.merge(left, right, on='key1')
```

In [ ]:
```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

Merging on Index

- Use the index as the join attribute

In [ ]:
```
left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
                   'value': range(6)})
right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
```

In [ ]:
```
left1
```

In [ ]:
```
right1
```

In [ ]:
```
# Equi-join on R.key=S.index
pd.merge(left1, right1, left_on='key', right_index=True)
```

In [ ]:
```
pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

In [ ]:
```
lefth = DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
                   'key2': [2000, 2001, 2002, 2001, 2002],
                   'data': np.arange(5.)})
righth = DataFrame(np.arange(12).reshape((6, 2)),
                   index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'],
                          [2001, 2000, 2000, 2000, 2001, 2002]],
                   columns=['event1', 'event2'])
lefth
```

In [ ]:
```
righth
```

In [ ]:
```
# Equi-Join on two pairs of columns
pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
```

In [ ]:
```
pd.merge(lefth, righth, left_on=['key1', 'key2'],
         right_index=True, how='outer')
```

In [ ]:
```
left2 = DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'],
                  columns=['Ohio', 'Nevada'])
right2 = DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
                   index=['b', 'c', 'd', 'e'], columns=['Missouri', 'Alabama'])
```

In [ ]:
```
left2
```

In [ ]:
```
right2
```

Using Other Functions to Perform Joins

- pd.merge(leftTable, rightTable, how, columns, ...)
- leftTable.join(otherTables, onCols, type..)

In [ ]:
```
pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

In [ ]:
```
left2.join(right2, how='outer')
```

In [ ]:
```
left1.join(right1, on='key')
```

In [ ]:
```
another = DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
                    index=['a', 'c', 'e', 'f'], columns=['New York', 'Oregon'])
```

In [ ]:
```
# join three tables
left2.join([right2, another])
```

In [ ]:
```
left2.join([right2, another], how='outer')
```

Concatenating Along An Axis

- NumPy has a concatenate() function that can append columns or append rows. Pandas provides concat() function
- By default, NumPy concatenate() put matrices side-by-side, aligned on rows
- Pandas concat() puts tables one on top of another, alligned on columns
  - This can be changed by setting axis=0 for concatenation vertically or axis=1 for concatenation horizontally

In [ ]:
```
arr = np.arange(12).reshape((3, 4))
arr
```

In [ ]:
```
# axis=0 concatenate rows, axis=1 concatenate columns
np.concatenate([arr, arr], axis=1)
```

In [ ]:
```
# Pandas concat method
# axis = 0 concatenate rows
s1 = Series([0, 1])
s2 = Series([2, 3, 4])
s3 = Series([5, 6])
pd.concat([s1, s2, s3])
```

In [ ]:
```
s1 = Series([0, 1], index=['a', 'b'])
s2 = Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = Series([5, 6], index=['f', 'g'])
pd.concat([s1, s2, s3])
```

In [ ]:
```
# axis =1 concatenate columns, but also align rows
pd.concat([s1, s2, s3], axis=1)
```

In [ ]:
```
s4 = pd.concat([s1 * 5, s3])
s4
```

In [ ]:
```
s1
```

In [ ]:
```
# Equivalent to outer equi-join on index
pd.concat([s1, s4], axis=1)
```

In [ ]:
```
# equivalent to equi-join on index
pd.concat([s1, s4], axis=1, join='inner')
```

In [ ]:
```
pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
```

In [ ]:
```
# Use hierarchical index to identify indexes from different serieses
result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
```

In [ ]:
```
result
```

In [ ]:
```
# Unstack the first level index (Much more on the unstack function later)
result.unstack()
```

In [ ]:
```
# Concatenate horizontally
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

In [ ]:
```
# Concatenate DataFrames
df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
                columns=['one', 'two'])
df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
                columns=['three', 'four'])
```

In [ ]:
```
df1
```

In [ ]:
```
df2
```

In [ ]:
```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

In [ ]:
```
# Can also pass input DataFrames in Dict
pd.concat({'level1': df1, 'level2': df2}, axis=1)
```

In [ ]:
```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
          names=['upper', 'lower'])
```

In [ ]:
```
# Concatenate by appending rows and making new index
df1 = DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
df2 = DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
```

In [ ]:
```
df1
```

In [ ]:
```
df2
```

In [ ]:
```
pd.concat([df1, df2], ignore_index=True)
```

In [ ]:
```
# Compare np.concatenate and pd.concat
df1 = DataFrame({'a': [1,2,3],
                 'b': [2, 3, 4]})
df2 = DataFrame({'b': [4, 5, 6],
                 'c': [2, 3, 5]})
np.concatenate([df1, df2], axis=1)
```

In [ ]:
```
pd.concat([df1, df2], axis=1)
```

Combining data with overlap

In [ ]:
```
a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
           index=['f', 'e', 'd', 'c', 'b', 'a'])
b = Series(np.arange(len(a), dtype=np.float64),
           index=['f', 'e', 'd', 'c', 'b', 'a'])
b[-1] = np.nan
```

In [ ]:
```
a
```

In [ ]:
```
b
```

In [ ]:
```
# In NumPy. If null value in a use the value in b
np.where(pd.isnull(a), b, a)
```

In [ ]:
```
# In Pandas, combine_first choose the values from the calling object whenever is p
ossible
b[:-2].combine_first(a[2:])
```

In [ ]:
```
df1 = DataFrame({'a': [1., np.nan, 5., np.nan],
                 'b': [np.nan, 2., np.nan, 6.],
                 'c': range(2, 18, 4)})
df2 = DataFrame({'a': [5., 4., np.nan, 3., 7.],
                 'b': [np.nan, 3., 4., 6., 8.]})
df1.combine_first(df2)
```

Reshaping and Pivoting

- Use the stack() and unstack() functions to change the shape and appearance of DataFrame
- Especially useful with hierarchical index

Reshaping with hierarchical indexing

In [ ]:
```
data = DataFrame(np.arange(6).reshape((2, 3)),
                 index=pd.Index(['Ohio', 'Colorado'], name='state'),
                 columns=pd.Index(['one', 'two', 'three'], name='number'))
data
```

In [ ]:
```
result = data.stack()
result
```

In [ ]:
```
result.unstack()
```

In [ ]:
```
result.unstack(0)
```

In [ ]:
```
result.unstack('state')
```

In [ ]:
```
s1 = Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
s2 = Series([4, 5, 6], index=['c', 'd', 'e'])
data2 = pd.concat([s1, s2], keys=['one', 'two'])
data2
```

In [ ]:
```
data2.unstack()
```

In [ ]:
```
# Unstack may generate null values, stack will ignore null values by default
data2.unstack().stack()
```

In [ ]:
```
data2.unstack().stack(dropna=False)
```

In [ ]:
```
df = DataFrame({'left': result, 'right': result + 5},
               columns=pd.Index(['left', 'right'], name='side'))
df
```

In [ ]:
```
df.unstack('state')
```

In [ ]:
```
df.unstack('state').stack('side')
```

Pivoting "long" to "wide" format

The columns of a data table may consist of values and dimensions. Some dimension may have a small set of items. A long format is the regular table format. A wide format may have several columns for the items in a dimension column.

For example: R(A, B) becomes R(A, B.b1, B.b2, B.b3) if column B contains three distinct values b1, b2, b3

In [ ]:
```
data = pd.read_csv('ch07/macrodata.csv')
periods = pd.PeriodIndex(year=data.year, quarter=data.quarter, name='date')
data = DataFrame(data.to_records(),
                 columns=pd.Index(['realgdp', 'infl', 'unemp'], name='item'),
                 index=periods.to_timestamp('D', 'end'))

ldata = data.stack().reset_index().rename(columns={0: 'value'})
wdata = ldata.pivot('date', 'item', 'value')
```

In [ ]:
```
# the long format
ldata[:10]
```

In [ ]:
```
# Use date as index, items as columns, values as elements
pivoted = ldata.pivot('date', 'item', 'value')
pivoted.head()
```

In [ ]:
```
# Suppose there are two columns of values
ldata['value2'] = np.random.randn(len(ldata))
ldata[:10]
```

In [ ]:
```
pivoted = ldata.pivot('date', 'item')
pivoted[:5]
```

In [ ]:
```
# Show only one column of value
pivoted['value'][:5]
```

In [ ]:
```
# Equivalently
unstacked = ldata.set_index(['date', 'item']).unstack('item')
unstacked[:7]
```

Data transformation

- Remove duplicate data
- Change data values by applying function
- Normalization
- Discritizaiton and Noise Smoothing
- Filling missing values
- Detect and replace outliers
- Data reduction using random sampling

Removing duplicates

In [ ]:
```
data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
                  'k2': [1, 1, 2, 3, 3, 4, 4]})
data
```

In [ ]:
```
data.duplicated()
```

In [ ]:
```
data.drop_duplicates()
```

In [ ]:
```
data['v1'] = range(7)
print(data)
data.drop_duplicates(['k1'])
```

In [ ]:
```
data.drop_duplicates(['k1', 'k2'], keep='last')
```

Transforming data using a function or mapping

A function or a composition of multiple functions can be applied to every element (or cell) in any slice of a DataFrame. The map() function does it automatically.

In [ ]:
```
data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami',
                           'corned beef', 'BACON', 'pastrami', 'honey ham',
                           'nova lox'],
                  'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

In [ ]:
```
# Want to add a column identify type of animal from which the type of meat is made
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

In [ ]:
```
# map applies a function to each element in the series
data['animal'] = data['food'].map(str.lower).map(meat_to_animal)
data
```

In [ ]:
```
# Apply a lambda function defined at the call
data['food'].map(lambda x: meat_to_animal[x.lower()])
```

Normalization

Normalization is to convert values from an arbitrary domain into a fixed domain, such as [0, 1], or [-2, 2], etc. There are several well-known normalizations.

- MiniMax Normalization: from $[min_A, max_A]$ to $[newmin_A, newmax_A]$
$$v' = \frac{v - min_A}{max_A - min_A}(newmax_A - newmin_A) + newmin_A$$
- Z-score Normalization: (μ: mean, σ: standard deviation):
$$v' = \frac{v - \mu_A}{\sigma_A}$$
- Decimal scaling:
$$v' = \frac{v}{10^j}$$
where j is the smallest integer such that $\max(|v'|) < 1$

In [ ]:
```
a = np.random.rand(20)
print("a =\n", a)
# MiniMax Normalization into [3, 10]
m = a.min()
M = a.max()
b = ((a-m)/(M-m))*(10-3)+3
print("b =\n", b)
# Z-score Normalization
mu = a.mean()
sd = a.std()
c = (a-mu)/sd
print("c =\n", c)
```

Replacing values

It may be neccessary to replace some outlier or missing values by other values. For this purpose, Pandas provides replace() and fillna() functions.

In [ ]:
```
data = Series([1., -999., 2., -999., -1000., 3.])
data
```

In [ ]:
```
data.replace(-999, np.nan)
```

In [ ]:
```
data.replace([-999, -1000], np.nan)
```

In [ ]:
```
data.replace([-999, -1000], [np.nan, 0])
```

In [ ]:
```
# Replacing based on a dict
data.replace({-999: np.nan, -1000: 0})
```

Filling Missing Values

In [ ]:
```
df = pd.DataFrame({'id': range(10),
                   'a' : [1.0, 3.0, np.nan, 4.0, 5.0, np.nan, np.nan, 4.0, 5.0, 3.
0],
                   'b' : ['red', None, 'blue', 'blue', None, None, 'gree', 'black'
, 'red', 'red']})
print(df)
df['a'] = df.a.fillna(df.a.mean())
df['b'] = df.b.fillna(df.b.mode()[0])
print(df)
```

Renaming axis indexes

In [ ]:
```
data = DataFrame(np.arange(12).reshape((3, 4)),
                 index=['Ohio', 'Colorado', 'New York'],
                 columns=['one', 'two', 'three', 'four'])
data
```

In [ ]:
```
data.index.map(str.upper)
```

In [ ]:
```
data.index = data.index.map(str.upper)
data
```

In [ ]:
```
# Change index and column case
data.rename(index=str.title, columns=str.upper)
```

In [ ]:
```
# Change index and column names
data.rename(index={'OHIO': 'INDIANA'},
            columns={'three': 'peekaboo'})
```

In [ ]:
```
# Always returns a reference to a DataFrame
_ = data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
data
```

Discretization and Noise Smoothing with Binning

- Discretization is to convert numeric data into categorical.
- Smoothing is to remove noise or extreem values
- Binning is a common method for discretization. Sort data values and place values into bins, then replace the values by its bin labels
- Binning can also be used to smooth data to reduce noises. For example, after binning, values in a column can be replaced by bin means or bin median, so that noises or outliers are smoothed out.
- Pandas provides two functions cut() and qcut() to perform binning

Equi-width binning using cut()

- **Equi-width Binning**: Divide the data range [min, max] equally into n sub-ranges. Assign each data to the bin according to its subrange. For example, we can divide the range (0, 100] into 4 bins: (0, 25], (25, 50], (50, 75], (75, 100]. The width of each bin is 25.
- cut(data, bins, labels): binning by range of value, can do equi-width binning
- For each input data, the output keeps the bin (either the label or the boundaries) for that data. The output is an object, also keeps the intervals of the bins

In [ ]:
```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

In [ ]:
```
# Set left/right boundaries of bins, and place data into bins
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
```

In [ ]:
```
# Find the bin id for each data value
cats.codes
```

In [ ]:
```
cats.describe
```

In [ ]:
```
cats.ravel
```

In [ ]:
```
pd.value_counts(cats)
```

In [ ]:
```
pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

In [ ]:
```
# assign different names to bins
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
pd.cut(ages, bins, labels=group_names)
```

In [ ]:
```
# equal-width bins with bin size automatically calculated from min/max values
data = np.random.rand(20)
print(data)
pd.cut(data, 4, precision=2)
```

Equi-depth binning using qcut()

- **Equi-depth Binning**: Divide the data range into bins of different width, so that, the bins contain the same number of data. For example, suppose there are 1000 data items. We can divide the range into 4 bins, so that, each bin contains 250 data items.
- qcut(data, quantile, labels): binning by quantile of the rank, can do equi-depth binning

In [ ]:
```
# Equal-depth bins where bins have equal numbers of values
data = np.random.randn(1000) # Normally distributed
cats = pd.qcut(data, 4) # Cut into quartiles
cats
```

In [ ]:
```
pd.value_counts(cats)
```

In [ ]:
```
pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
```

In [ ]:
```
pd.value_counts(pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.]))
```

Exercise: replace each input value by the mean of its bin

Detecting and filtering outliers

- Identify outlier (or extreem values)
  - check statistics
  - view boxplot
  - test using threshold values
- Once identified, outliers can be either removed or replaced

In [ ]:
```
np.random.seed(12345)
data = DataFrame(np.random.randn(1000, 4))
data.describe()
```

In [ ]:
```
data
```

In [ ]:
```
# Find in column 3, values with absolute value greater than 3
col = data[3]
col[np.abs(col) > 3]
```

In [ ]:
```
np.abs(data) > 3
```

In [ ]:
```
(np.abs(data) > 3).any(1)
```

In [ ]:
```
# Select rows that contains a value >3 or <-3
data[(np.abs(data) > 3).any(1)]
```

In [ ]:
```
# Cap the values by -3 and +3
data[np.abs(data) > 3] = np.sign(data) * 3
data.describe()
```

In [ ]:
```
data[(np.abs(data) == 3).any(1)]
```

Permutation and Random Sampling

- One way to reduce data volume is by random sampling.
- Pandas provides several functions to take random samples
  - permutaiton() randomly reorder values in a series or rows in a DataFrame. This function can be used together with take() function to get random samples from DataFrames.
  - another function is sample()
- There are several ways to take random samples.
  - **Random sampling without replacement**: Take n random samples, and no two samples can be same.
  - **Random sampling with replacement**: Take n samples randomly and allow the same sample to be selected multiple times.

In [ ]:
```
df = DataFrame(np.arange(5 * 4).reshape((5, 4)))
df
```

In [ ]:
```
# Get a permutaion of [0, 1, 2, 3, 4]
sampler = np.random.permutation(5)
sampler
```

In [ ]:
```
# take rows from a DataFrame in the given order
df.take(sampler)
```

In [ ]:
```
# take randomly selected 3 rows without replacement
df.take(np.random.permutation(len(df))[:3])
```

In [ ]:
```
# or use the DataFrame.sample() function
df.sample(n=3)
```

In [ ]:
```
# take a random sample of 10 with replacement
bag = np.array([5, 7, -1, 6, 4])
bag
```

In [ ]:
```
sampler = np.random.randint(0, len(bag), size=10)
sampler
```

In [ ]:
```
draws = bag.take(sampler)
draws
```

In [ ]:
```
# Another example
df.sample(n=3, replace=True)
```

Additional Topics

Computing indicator / dummy variables

In [ ]:
```
df = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                'data1': range(6)})
df
```

In [ ]:
```
# Which row contains a, b, c in column key?
pd.get_dummies(df['key'])
```

In [ ]:
```
dummies = pd.get_dummies(df['key'], prefix='key')
df_with_dummy = df[['data1']].join(dummies)
df_with_dummy
```

In [ ]:
```
mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('../week02/ch02/movielens/movies.dat', sep='::', header=Non
e,
                        names=mnames)
movies[:10]
```

In [ ]:
```
genre_iter = (set(x.split('|')) for x in movies.genres)
genres = sorted(set.union(*genre_iter))
genres
```

In [ ]:
```
# Create a table filled with zeros for all genres
dummies = DataFrame(np.zeros((len(movies), len(genres))), columns=genres)
dummies
```

In [ ]:
```
# for each movie, set each genres to 1 in the dummies table
for i, gen in enumerate(movies.genres):
    dummies.ix[i, gen.split('|')] = 1
dummies
```

In [ ]:
```
movies_windic = movies.join(dummies.add_prefix('Genre_'))
movies_windic.iloc[:3]
```

In [ ]:
```
# Combine get_dummies() and cut() functions to get an statistical indicator for da
ta
np.random.seed(12345)
```

In [ ]:
```
values = np.random.rand(10)
values
```

In [ ]:
```
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
pd.get_dummies(pd.cut(values, bins))
```

String manipulation

String object methods

In [ ]:
```
val = 'a,b,  guido'
val.split(',')
```

In [ ]:
```
pieces = [x.strip() for x in val.split(',')]
pieces
```

In [ ]:
```
first, second, third = pieces
first + '::' + second + '::' + third
```

In [ ]:
```
'::'.join(pieces)
```

In [ ]:
```
'guido' in val
```

In [ ]:
```
val.index(',')
```

In [ ]:
```
val.find(':')
```

In [ ]:
```
# index() is like find(), but raise ValueError when the substring is not found.
# val.index(':')
```

In [ ]:
```
val.count(',')
```

In [ ]:
```
val.replace(',', '::')
```

In [ ]:
```
val.replace(',', '')
```

Regular expressions

In [ ]:
```
import re
text = "foo    bar\t baz  \tqux"
re.split('\s+', text)
```

In [ ]:
```
# compile a regular expression as a pattern
regex = re.compile('\s+')
regex.split(text)
```

In [ ]:
```
regex.findall(text)
```

In [ ]:
```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
```

In [ ]:
```
# Define a pattern for email address
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

In [ ]:
```
regex.findall(text)
```

In [ ]:
```
# search returns the first match any where in a line
m = regex.search(text)
m
```

In [ ]:
```
text[m.start():m.end()]
```

In [ ]:
```
# Match only matches from the beginning of the text, not where else
print(regex.match(text))
```

In [ ]:
```
# sub() replace the first group by the string in the text
print(regex.sub('REDACTED', text))
```

In [ ]:
```
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
regex = re.compile(pattern, flags=re.IGNORECASE)
```

In [ ]:
```
m = regex.match('wesm@bright.net')
m.groups()
```

In [ ]:
```
# returns list of tuples representing the matched groups
regex.findall(text)
```

In [ ]:
```
print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
```

In [ ]:
```
# Assign names to pattern groups
regex = re.compile(r"""
    (?P<username>[A-Z0-9._%+-]+)
    @
    (?P<domain>[A-Z0-9.-]+)
    \.
    (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)
```

In [ ]:
```
m = regex.match('wesm@bright.net')
m.groupdict()
```

Vectorized string functions in pandas

Pandas Series uses a sub() function to apply string operations on each data item, and can deal with NA values

In [ ]:
```
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
        'Rob': 'rob@gmail.com', 'Wes': np.nan}
data = Series(data)
```

In [ ]:
```
data
```

In [ ]:
```
data.isnull()
```

In [ ]:
```
data.str.contains('gmail')
```

In [ ]:
```
pattern
```

In [ ]:
```
data.str.findall(pattern, flags=re.IGNORECASE)
```

In [ ]:
```
matches = data.str.match(pattern, flags=re.IGNORECASE)
matches
```

In [ ]:
```
matches.str.get(1)
```

In [ ]:
```
matches.str[0]
```

In [ ]:
```
data.str[:5]
```

Example: USDA Food Database

{ "id": 21441, "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY, Wing, meat and skin with breading", "tags": ["KFC"], "manufacturer": "Kentucky Fried Chicken", "group": "Fast Foods", "portions": [{ "amount": 1, "unit": "wing, with skin", "grams": 68.0 }, ...], "nutrients": [{ "value": 20.8, "units": "g", "description": "Protein", "group": "Composition" }, ...] }

Each food has a number of identifying attributes along with two lists of nutrients and portion sizes. Having the data in this form is not particularly amenable for analysis, so we need to do some work to wrangle the data into a better form.

In [ ]:
```
import json
db = json.load(open('ch07/foods-2011-10-03.json'))
len(db)
```

In [ ]:
```
db[:2]
```

In [ ]:
```
# db[0] is the entire file
db[0].keys()
```

In [ ]:
```
db[0]['nutrients'][0]
```

In [ ]:
```
# Load the nutrients of the first food into a DataFrame
nutrients = DataFrame(db[0]['nutrients'])
nutrients[:7]
```

In [ ]:
```
# Load some remaining info into another DataFrame
info_keys = ['description', 'group', 'id', 'manufacturer']
info = DataFrame(db, columns=info_keys)
```

In [ ]:
```
info[:5]
```

In [ ]:
```
info
```

In [ ]:
```
# Counting foods by their food groups
pd.value_counts(info.group)[:10]
```

In [ ]:
```
# Build a long table of nutrients for all foods
nutrients = []

# Build a DataFrame of nutrients for each food
for rec in db:
    fnuts = DataFrame(rec['nutrients'])
    fnuts['id'] = rec['id']
    nutrients.append(fnuts)

# Concatenate into a long table
nutrients = pd.concat(nutrients, ignore_index=True)
```

In [ ]:
```
nutrients
```

In [ ]:
```
# Drop duplicate
nutrients.duplicated().sum()
```

In [ ]:
```
nutrients = nutrients.drop_duplicates()
```

In [ ]:
```
# Renaming columns to make specific meanings of each column
col_mapping = {'description' : 'food',
               'group'       : 'fgroup'}
info = info.rename(columns=col_mapping, copy=False)
info
```

In [ ]:
```
col_mapping = {'description' : 'nutrient',
               'group' : 'nutgroup'}
nutrients = nutrients.rename(columns=col_mapping, copy=False)
nutrients
```

In [ ]:
```
# join the two tables
ndata = pd.merge(nutrients, info, on='id', how='outer')
```

In [ ]:
```
ndata
```

In [ ]:
```
ndata.ix[30000]
```

In [ ]:
```
# Plot the histogram of food groups
result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)
result['Zinc, Zn'].sort_values().plot(kind='barh')
```

In [ ]:
```
# Find which food is most dense in each nutrient
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.xs(x.value.idxmax())
get_minimum = lambda x: x.xs(x.value.idxmin())

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# make the food a little smaller
max_foods.food = max_foods.food.str[:50]
```

In [ ]:
```
# Example for Amino Acids
max_foods.loc['Amino Acids']['food']
```